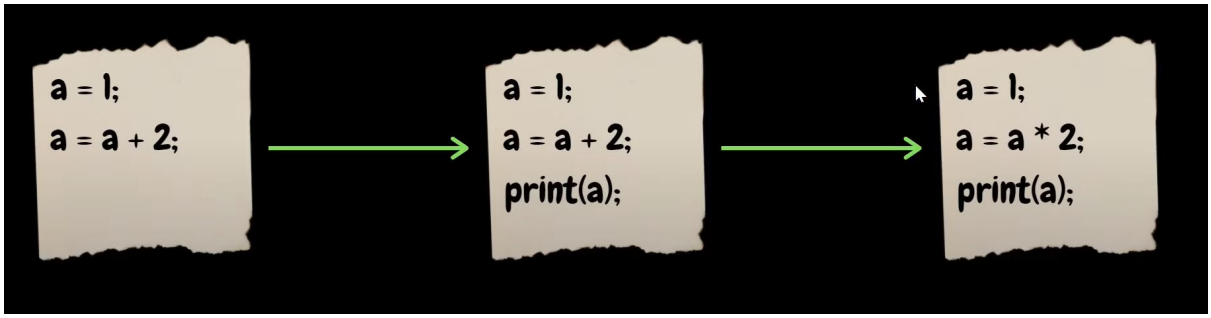
1. Git Repository

A Git repository (often abbreviated as "repo") is a storage space where your project's files and the entire revision history (or version history) of these files are stored. Git is a version control system, and a repository is the core unit where all the changes, versions, and branches of a project are managed.

2. Version Control System

Keep track of all the changes done



Two Types:

1. **Centralized Version Control System**

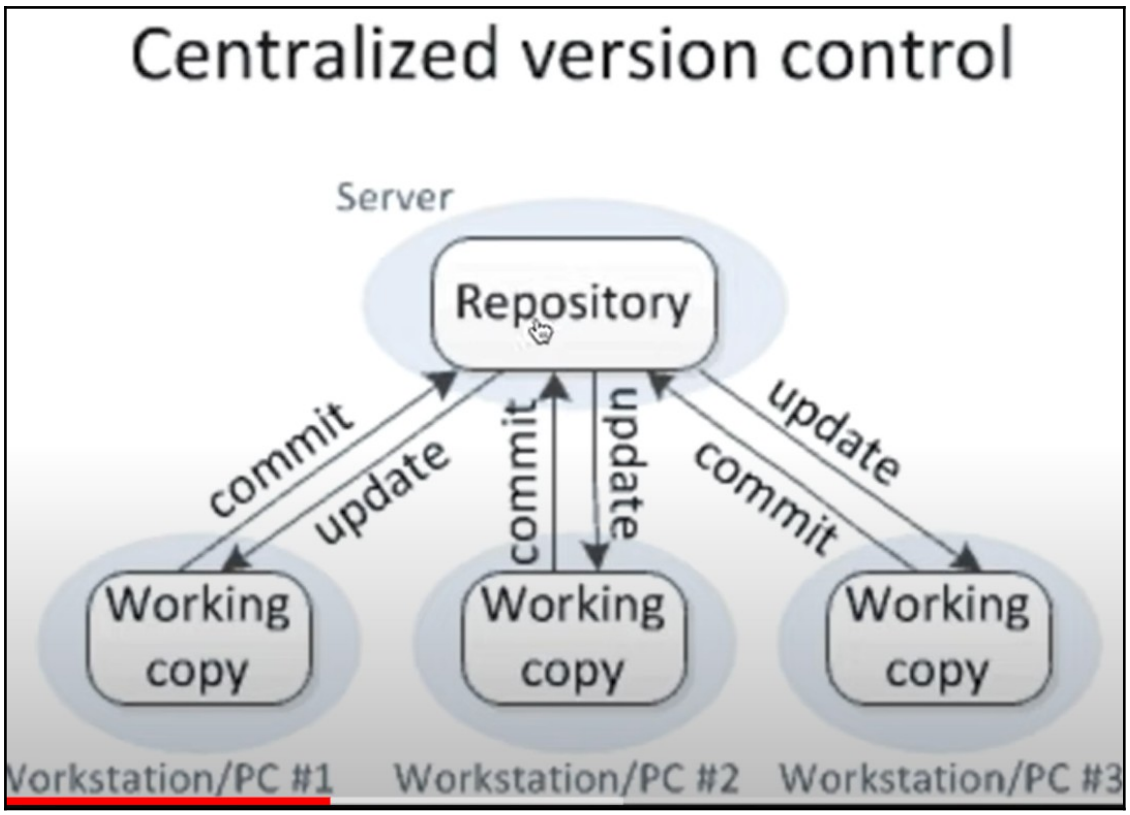
   Example:

   a. Subversion
   b. MS Team Foundation Server
   c. CVS (Concurrent Version System)

2. **Distributed Version Control System**

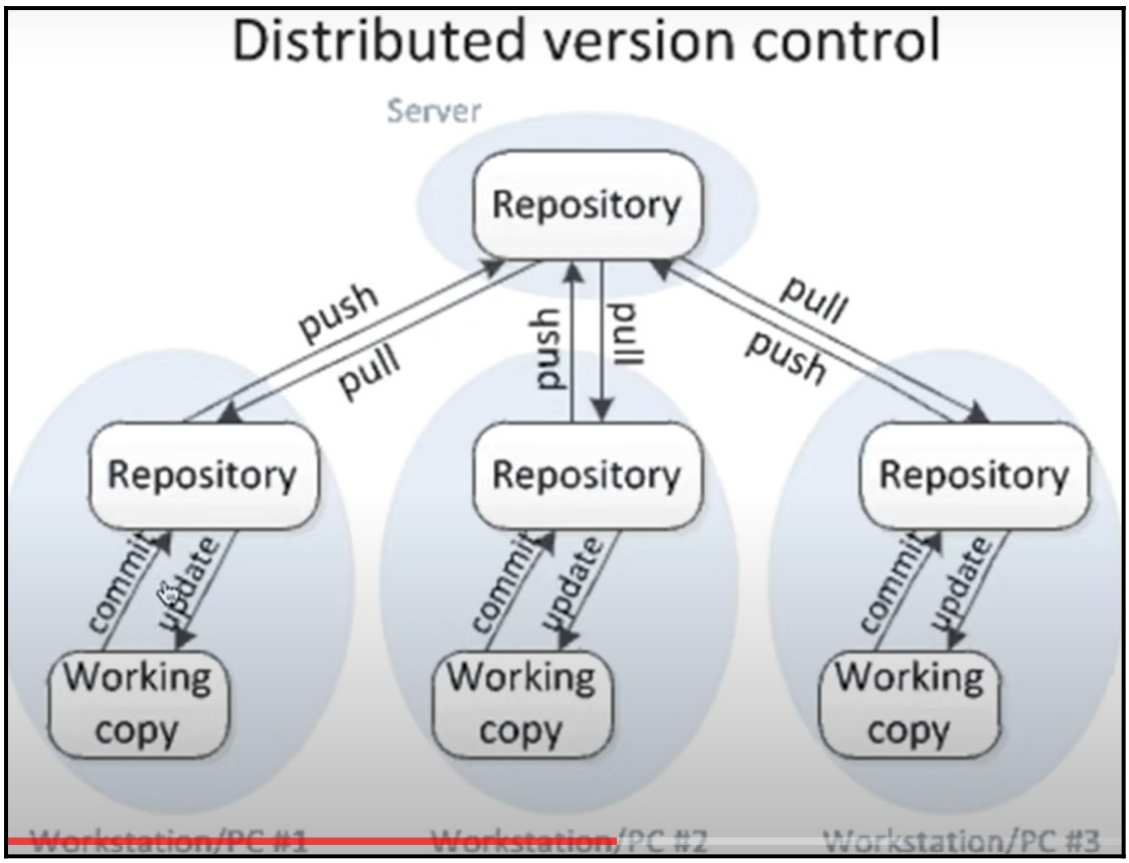
   Example:

   a. Git
   b. Mercurial



Please connect with me via below social media

Instagram: https://www.instagram.com/rba_infotech/
LinkedIn:
https://www.linkedin.com/company/70977316/admin/dashboard/
Facebook:https://www.facebook.com/people/RBA-Infotech/100043552406579/



3. Git Commands

Complete Git Flow

1. Create repository in Github
2. Open gitbash application from current directory
3. git config --global user.email senthil.natarajan@rbainfotech.com
4. git config --global user.name "RBA-Infotech"
5. git init (only once)
6. Create some files here
7. Git status
8. Git add <filename> or git add .
9. Git status
10. Git restore - to get tracked files, but not commited

Please connect with me via below social media

Instagram: https://www.instagram.com/rba_infotech/
LinkedIn:
https://www.linkedin.com/company/70977316/admin/dashboard/
Facebook:https://www.facebook.com/people/RBA-Infotech/100043552406579/

11. git commit -m "addition file1, file2" (-m – add commit message)
12. git push origin master
13. first time coping files - git clone <git url> , from second time onwards, git pull -r
14. pull request
15. merge request
16. clone the git repository – git clone <.git>
17. create breach
18. git branch
19. git checkout <branch name>
20. create file
21. git status
22. git add <filename>
23. git status
24. git commit -m “ ”

1. git init
2. git status
3. git add .
4. git status
5. git commit -m “added CANS files”
6. git remote add origin <git url of repository>
7. git push -u origin master
8.

- To see all commit logs - git log
- To display the log in one line - git log –oneline
- to move Head position to any commit - git reset –hard or --soft <revision id> -
  by default is - --soft
- To delete any in-between commits - git revert <revision id>
- To create branch – git branch <branch name>
- To switch branch – git checkout <target branch name>
- To delete branch – git branch -d <branch name>

- To rebase branch – git rebase <from branch> <target branch>
- To bring only specific commit – git cherry-pick <Revision id>
- To stash few changes – git stash
- To see all stash list – git stash list
- To bring stashed changes back – git stash pop <pop id (optional)>
- To squash many commits into single commit – squash is used
-

4. Key Components of a Git Repository

1. Working Directory:

The working directory is where you work on your project files. Any changes made here can be tracked by Git.

2. Staging Area (Index):

Before committing changes to the repository, files are placed in the staging area. This is like a buffer where you prepare changes for the next commit. Only files that are staged will be included in the next commit.

3. Commit:

A commit is a snapshot of your project at a specific point in time. It includes all the changes that were staged and records them in the repository's history. Each commit has a unique identifier (a SHA-1 hash) and typically includes a commit message describing the changes.

4. Branches:

Branches are pointers to specific commits in the repository. The default branch is usually called `main` or `master`. Branching allows you to diverge from the main line of development to work on features or fixes in isolation.

5. Remote Repositories:

A Git repository can be hosted on a remote server (like GitHub, GitLab, Bitbucket). Remote repositories allow multiple collaborators to work on the same project by pushing (uploading) and pulling (downloading) changes.

6. .git Directory:

This is a hidden directory located at the root of every Git repository. It contains all the necessary metadata and object database for your project, including commits, branches, and tags.

7. Origin

In Git, "origin" is a *remote* repository. More specifically, it's a *shorthand name* that Git uses for the remote repository that you originally cloned from. It's the default name, but it's just a convention; you could name it something else if you wanted

8. Head

HEAD is a reference that points to the current commit on the current branch. Think of it as a pointer that keeps track of where you are in your project's history.

5. Types of Git Repositories:

1. Local Repository:

A Git repository on your local machine.

2. Remote Repository:

A repository hosted on a remote server, accessible by other collaborators.

Git repositories are fundamental for managing code changes, collaborating with others, and maintaining a project’s history. They enable developers to track changes, revert to previous states, and work together on complex projects without conflicts.

6. Common Git Repository Operations:

1. Cloning: Copying a remote repository to your local machine with `git clone <repository_url>`.
2. Committing: Saving your changes to the repository with `git commit -m "Commit message"`.
3. Pushing: Uploading local commits to a remote repository with `git push`.
4. Pulling: Downloading changes from a remote repository to your local machine with `git pull`.
5. Branching: Creating a new branch with `git branch <branch_name>` and switching to it with `git checkout <branch_name>`.
6. Merging: Combining changes from different branches with `git merge <branch_name>`.

1. Pushing files from local system to GitHub Repository

1. Open Git Bash from the project that you want to push from File Explorer
2. git config --global user.email "senthil.natarajan@rbainfotech.com"
3. git config --global user.name "RBA-Infotech"
4. git init (only once)
5. git status
6. git add .
7. git status
8. git commit -m "first commit"
9. git remote add origin https://github.com/selenium56sen/Jenkin_GitHup_Flipkart.git (only one time enough)_
10. git push -u origin master
11. check the github and refresh, now you can find all the files

Please connect with me via below social media

Instagram: https://www.instagram.com/rba_infotech/
LinkedIn:
https://www.linkedin.com/company/70977316/admin/dashboard/
Facebook:https://www.facebook.com/people/RBA-Infotech/100043552406579/

2. Git Commands for Step by Step procedure for from start such as clone repository, then team member creating branch, then creating files, merging the files with Remote Repository and raise Pull request

Complete Git Flow

1. Create repository in Github
2. Open gitbash application from current directory
3. git config --global user.email senthil.natarajan@rbainfotech.com
4. git config --global user.name "RBA-Infotech"
5. git init
6. Create some files here
7. Git status
8. Git add <filename> or git add .
9. Git status
10. Git restore - to get tracked files, but not commited
11. git commit -m "addition file1, file2" (-m – add commit message)
12. git push origin master
13. first time coping files - git clone <git url> , from second time onwards, git pull -r
14. pull request
15. merge request
16. clone the git repository – git clone <.git>
17. create branch
18. git branch
19. git checkout <branch name>
20. create file
21. git status
22. git add <filename>
23. git status

24. git commit -m “ ”

3. Other Git Commands

1. git log

Git log: It displays the commit history of a repository

This will display the commit history in a default format, showing each commit's:

- Commit hash (SHA-1 checksum)
- Author
- Date and time
- Commit message

- Git log – oneline
- git log --stat
- git log --oneline –graph
- git log --oneline –decorate
- git log --oneline –all

2. git revert

git revert is a Git command used to undo changes made by a specific commit. Unlike git reset, which moves the branch pointer backward and can rewrite history (especially if you've already pushed your commits), git revert creates a *new commit* that *reverses* the changes introduced by the target commit. This preserves the project's history, making it a safer option for undoing changes that have already been shared with others.

How It Works

- Git creates a new commit that reverses the changes introduced by the specified commit.
- The specified commit remains in the history.

```
git revert abc1234
```

- This will open a text editor to confirm or modify the commit message for the revert commit.
- Save and close the editor to finalize the revert.

3. git reset

git reset is a Git command used to undo changes in your repository. It has three main modes: **soft**, **mixed**, and **hard**, each affecting your commit history, staging area, and working directory differently.

4. Syntax

git reset [<mode>] <commit>

- <mode>: Specifies how the command will behave (--soft, --mixed, --hard).
- <commit>: The commit hash or reference you want to reset to.

---

5. Modes of git reset

1. --soft

- **Effect**: Moves the HEAD pointer to the specified commit but keeps the changes staged.
- **Use Case**: Undo commits but keep changes staged for editing or committing again.

git reset --soft <commit>

Example:

git reset --soft HEAD~1

- Undo the latest commit, keeping changes in the staging area.

---

2. --mixed (Default mode)

- **Effect**: Moves the HEAD pointer to the specified commit and unstages changes, leaving them in the working directory.

- **Use Case**: Undo commits and unstage changes for further edits.

git reset --mixed <commit>

Example:

git reset --mixed HEAD~1

- Undo the last commit and move changes to the working directory.

---

3. --hard

- **Effect**: Moves the HEAD pointer to the specified commit and discards all changes in the working directory and staging area.
- **Use Case**: Completely reset the repository to a previous state (use with caution, as changes will be lost).

git reset --hard <commit>

Example:

git reset --hard HEAD~1

- Completely erase the latest commit and its changes.

Examples

1. **Undo the Last Commit and Keep Changes Staged**

bash

Copy code

git reset --soft HEAD~1

2. **Unstage Changes Without Removing Them**

bash

Copy code

git reset --mixed HEAD

3. **Completely Discard Changes**

bash

Copy code

git reset --hard HEAD

4. **Reset to a Specific Commit**

bash

Copy code

```
git reset --mixed <commit_hash>
```

Reset vs. Revert

- **git reset**: Alters commit history and can remove commits. Ideal for local/private branches.
- **git revert**: Adds a new commit that undoes a previous one. Preferred for shared branches.

Key Differences between git revert and git reset:

Feature	git revert	git reset
Action	Creates a new commit that undoes changes	Moves the branch pointer backward
History	Preserves history	Can rewrite history (if used with --hard)
Safety	Safer for shared branches	Potentially dangerous for shared branches
Use Case	Undo changes that have already been pushed	Undo local changes that haven't been pushed

4. git rebase

git rebase is a Git command used to reapply commits on top of another base commit. It is commonly used to:

1. Keep a feature branch up-to-date with the main branch.
2. Rewrite commit history to make it cleaner and more linear.

---

5. How Rebase Works

Rebasing moves a series of commits from their current base to a new base, essentially replaying those commits on top of the new base commit.

For example:

Before Rebase:

```
css
```

Copy code

```
A --- B --- C  (main)
       \
        D --- E (feature)
```

After Rebase (feature rebased onto main):

mathematica

Copy code

```
A --- B --- C --- D' --- E'  (feature)
```

Commits D and E are reapplied as D' and E' with new commit hashes.

---

6. Basic Syntax

git rebase <base-branch>

---

7. Rebase Use Cases

1. Rebase a Feature Branch onto Main

Keeps your feature branch up-to-date with changes in the main branch.

bash

Copy code

```
git checkout feature
git rebase main
```

Steps:

- Git replays the commits from feature on top of the latest main.
- Resolve conflicts if any arise.
- Continue the rebase after resolving conflicts:

bash

Copy code

```
git rebase --continue
```

---

2. Interactive Rebase

Allows you to edit, squash, or reorder commits.

bash

Copy code

git rebase -i <base-commit>
Example:
bash
Copy code
git rebase -i HEAD~3
Steps:
1. A list of commits appears in your editor:
sql
Copy code
pick abc123 Commit message 1
pick def456 Commit message 2
pick ghi789 Commit message 3
2. Modify actions (pick, squash, edit, etc.).
3. Save and close the editor.
4. Follow Git's instructions for completing the rebase.

---

3. Abort a Rebase

If something goes wrong, you can cancel the rebase and return to the original state:

bash
Copy code
git rebase --abort

---

4. Skip a Commit

If a commit cannot be applied (e.g., due to conflicts) and you want to ignore it:

bash
Copy code
git rebase --skip

5. git merge

git merge is a Git command used to combine changes from one branch into another. It integrates the work from different branches, creating a unified history.

---

6. Basic Syntax

bash

Copy code

git merge <branch-name>

- **<branch-name>**: The branch you want to merge into the current branch.

---

7. Types of Merges

1. Fast-Forward Merge

Occurs when the current branch is directly ahead of the branch being merged. Git simply moves the branch pointer forward.

Example:

bash

Copy code

git checkout main

git merge feature

Before:

css

Copy code

A --- B --- C (main)

\

D --- E (feature)

After (fast-forward):

css

Copy code

A --- B --- C --- D --- E (main)

2. Three-Way Merge

Occurs when the branches have diverged. Git creates a new merge commit to combine the changes.

Example:

bash

Copy code

git checkout main

git merge feature

Before:
css
Copy code
A --- B --- C (main)
\
D --- E (feature)
After (three-way merge):
css
Copy code
A --- B --- C --- F (main)
\ /
D --- E

---

8. Steps to Perform a Merge

1. Update the Target Branch

Ensure the branch you want to merge into is up-to-date:
bash
Copy code
git checkout main
git pull origin main

2. Merge the Feature Branch

Merge the branch into the current branch:
bash
Copy code
git merge feature

3. Resolve Conflicts (if any)

If there are merge conflicts, Git will pause the merge and mark the files with conflicts. Resolve them manually:

- Edit the conflicted files.
- Mark the conflicts as resolved:

bash
Copy code
git add <file>

- Continue the merge:

bash
Copy code
git commit
or
bash
Copy code
git merge --continue

4. Push the Changes

After a successful merge, push the changes:

bash
Copy code
git push origin main

Merge vs Rebase

Aspect	Merge	Rebase
History	Retains all commits, including merge commits.	Creates a linear history.
Commit Integrity	Doesn't modify commit history.	Rewrites commit history.
Conflict Resolution	Happens once during the merge.	Happens at each commit being replayed.
Use Case	Suitable for preserving history of branches.	Suitable for creating a clean history.

6. fork

"forking" refers to creating a *copy* of a repository under your own account. It's like taking a snapshot of the project at a specific point in time.

Here's a breakdown:

What happens when you fork a repository:

1. **Copying the Repository:** A complete copy of the original repository (including all branches, commits, and tags) is created in your account. This copy is independent of the original repository.

2. **Remote Relationship:** Your forked repository has a remote connection back to the original repository. This allows you to fetch updates from the original repository. The original repository is often referred to as the "upstream" repository.
3. **No Direct Push Access:** You typically do *not* have direct write access (push access) to the original repository unless you are explicitly granted it by the owner.

Why fork a repository?

- **Contributing to Open Source Projects:** Forking is the standard way to contribute to open-source projects on platforms like GitHub. You fork the project, make your changes in your forked copy, and then submit a "pull request" (or "merge request" on GitLab/Bitbucket) to the original project maintainers.
- **Experimenting with Code:** You can fork a repository to experiment with its code without affecting the original project. You can make changes, try out new features, or even break things without worrying about breaking the original project for others.
- **Starting Your Own Project Based on Existing Code:** You can fork a repository as a starting point for your own project. You can then modify the code to suit your needs.

Workflow for Contributing to an Open Source Project (using GitHub as an example):

1. **Fork the repository:** On the project's GitHub page, click the "Fork" button. This creates a copy of the repository in your GitHub account.
2. **Clone your fork:** Clone the forked repository to your local machine:

Bash

```
git clone https://github.com/your_username/forked_repository.git
```

3. **Create a branch:** Create a new branch for your changes:

Bash

```
git checkout -b my-feature-branch
```

4. **Make your changes:** Make your code changes, commit them locally:

Bash

```
# ... make changes ...
git add .
git commit -m "Describe your changes"
```

5. **Push your changes to your fork:** Push your local branch to your forked repository on GitHub:

Bash

```
git push origin my-feature-branch
```

6. **Create a pull request:** On the GitHub page of your forked repository, click the "Compare & pull request" button. This will open a pull request to the original repository.
7. **Review and discussion:** The project maintainers will review your pull request, and there may be some discussion or requests for changes.
8. **Merge (or rejection):** If the maintainers approve your changes, they will merge your pull request into the original repository. If not, they might suggest further changes or decline the pull request.

7. git clone

git clone is a Git command used to create a local copy of a remote repository. It downloads the entire repository, including its history, branches, and files, to your local machine.

---

8. Basic Syntax

bash

Copy code

```
git clone <repository-url> [directory]
```

- **<repository-url>**: The URL of the remote repository you want to clone.
- **[directory]**: (Optional) The name of the directory where the repository will be cloned. If not provided, Git uses the repository name by default.

---

9. How It Works

When you run git clone:

1. Git downloads all the files and commit history of the specified repository.
2. It sets up a connection to the remote repository as origin.
3. The default branch (usually main or master) is checked out.

---

10. Examples

1. Clone a Repository

To clone a repository:

bash

Copy code

```
git clone https://github.com/example-user/example-repo.git
```

This creates a directory named example-repo with the repository contents.

2. Clone a Repository into a Specific Directory

bash

Copy code

```
git clone https://github.com/example-user/example-repo.git my-folder
```

This clones the repository into a folder named my-folder.

3. Clone a Specific Branch

To clone only a specific branch:

bash

Copy code

```
git clone --branch <branch-name> https://github.com/example-user/example-repo.git
```

Example:

bash

Copy code

```
git clone --branch dev https://github.com/example-user/example-repo.git
```

4. Clone a Repository with Submodules

If the repository contains submodules:

bash

Copy code

git clone --recurse-submodules https://github.com/example-user/example-repo.git

5. Shallow Clone (Only Recent History)

To clone a repository with limited commit history (for faster downloads):

bash

Copy code

git clone --depth 1 https://github.com/example-user/example-repo.git

---

11. Checking Cloned Repository Information

1. **List Remote Repositories** After cloning, Git automatically adds the remote repository under the name origin:

bash

Copy code

git remote -v

Example output:

perl

Copy code

origin https://github.com/example-user/example-repo.git (fetch)

origin https://github.com/example-user/example-repo.git (push)

2. **View Branches**

bash

Copy code

git branch -a

Lists all branches, including remote ones.

---

12. Common Options

1. **--branch <branch>** Clone a specific branch.
2. **--depth <number>** Perform a shallow clone with a limited commit history.
3. **--recurse-submodules** Clone the repository and its submodules.
4. **--single-branch** Clone only the specified branch, even if others exist.

---

13. Workflow Example

Step 1: Clone a Repository

bash

Copy code

```
git clone https://github.com/example-user/example-repo.git
```

Step 2: Navigate into the Directory

bash

Copy code

```
cd example-repo
```

Step 3: Check the Current Branch

bash

Copy code

```
git branch
```

Step 4: Make Changes and Push

- Make changes to the repository files.
- Add, commit, and push changes:

bash

Copy code

```
git add .
git commit -m "My changes"
git push origin main
```

---

14. Cloning Private Repositories

If the repository is private, you'll need access credentials (username/password, SSH keys, or token):

bash

Copy code

```
git clone https://<username>@github.com/example-user/private-repo.git
```

Would you like help cloning a repository or setting up SSH keys for easier access?

8. git push -u origin master

The -u option in the git push command is short for **--set-upstream**. It is used to establish a tracking relationship between your local branch and a remote branch, making future git push and git pull commands simpler.

---

9. Understanding git push -u

When you use git push -u origin master:

1. Git pushes your local master branch to the remote repository origin.
2. Git sets the upstream branch for your local master branch to the remote origin/master.
3. Future git push or git pull commands can be used without specifying the branch name, as the upstream branch is already set.

---

10. Syntax

bash

Copy code

git push -u <remote-name> <branch-name>

- **-u**: Sets the upstream branch for the specified local branch.
- **<remote-name>**: Typically origin, which is the default name for the remote repository.
- **<branch-name>**: The name of your local branch.

---

11. Why Use -u?

1. **Simplifies Workflow**: After setting the upstream branch, you can use:

bash

Copy code

git push

instead of:

bash

Copy code

git push origin master

2. **Links Local and Remote Branches**: It enables Git to automatically determine which remote branch to push to or pull from.
3. **Convenience**: When pulling changes:

bash
Copy code
git pull

Git will fetch updates from the upstream branch without needing to specify origin or master.

---

12. Example

Step 1: Push and Set Upstream

bash
Copy code
git push -u origin main

Step 2: Push or Pull Changes in the Future

Once the upstream branch is set, you can simply run:

bash
Copy code
git push

or

bash
Copy code
git pull

---

13. Checking Upstream Branch

To see the upstream branch for your local branch:

bash
Copy code
git branch -vv

Output example:

css
Copy code
* main abc123 [origin/main] Initial commit

The [origin/main] indicates that the local main branch is tracking the remote origin/main.

---

14. Changing or Removing Upstream Branch

1. **Change the Upstream Branch**:

bash
Copy code
git branch --set-upstream-to=<remote>/<branch>
Example:
bash
Copy code
git branch --set-upstream-to=origin/dev

2. **Remove the Upstream Branch**:

bash
Copy code
git branch --unset-upstream

---

15. Precautions

1. **Correct Spelling**: Ensure the command uses origin, not origni, and master, not maste.

bash
Copy code
git push -u origin master

2. **Use -u Only Once**: The -u flag is needed only during the first push to set the upstream branch. Afterward, you can omit it.

9. Pull Request (PR)

A **pull request (PR)** is a feature in Git hosting platforms like GitHub, GitLab, and Bitbucket that allows you to propose changes to a repository. It serves as a collaborative way to review and merge code changes into a target branch, typically the main branch of the repository.

---

10. How a Pull Request Works

1. **Fork or Clone the Repository**: Create a copy of the repository if you don't have write access.

2. **Create a New Branch**: Make your changes in a separate branch rather than directly on the main branch.
3. **Commit and Push Changes**: Push the changes from your local branch to the remote repository.
4. **Open a Pull Request**: Use the Git hosting platform to propose merging your branch into the target branch of the original repository.
5. **Code Review and Discussion**: The repository maintainers or collaborators review the proposed changes. They may request edits or approve the PR.
6. **Merge the Changes**: Once approved, the PR is merged into the target branch.

---

11. Creating a Pull Request

Step 1: Create a Branch

Create a new branch for your changes:

bash

Copy code

```
git checkout -b feature-branch
```

Step 2: Make Changes

Edit the files and commit your changes:

bash

Copy code

```
git add .
git commit -m "Add feature or fix issue"
```

Step 3: Push the Changes

Push the branch to the remote repository:

bash

Copy code

```
git push origin feature-branch
```

Step 4: Open the Pull Request

1. Go to the Git hosting platform (e.g., GitHub).
2. Navigate to your forked repository.
3. Click **New Pull Request**.

4. Choose the base branch (e.g., main) and compare it with your branch (feature-branch).
5. Add a title and description explaining your changes.
6. Submit the pull request.

---

12. Best Practices for Pull Requests

1. **Create Small, Focused PRs**: Keep PRs small and focused on a single feature or fix for easier review.
2. **Write Clear Descriptions**: Clearly explain what the PR does, why the changes are needed, and how they were implemented.
3. **Follow Contribution Guidelines**: Adhere to the repository's guidelines, including coding standards and commit message conventions.
4. **Test Your Changes**: Ensure your changes work as expected and do not introduce bugs.
5. **Respond to Feedback**: Be open to suggestions and make the necessary changes promptly.

---

13. Merging Pull Requests

Once a pull request is approved, it can be merged into the target branch. Merge methods depend on the platform and configuration:

1. **Merge Commit**: Combines all commits from the feature branch into a single commit on the main branch.
2. **Squash and Merge**: Squashes all commits from the feature branch into one before merging.
3. **Rebase and Merge**: Rebases the commits from the feature branch onto the main branch and merges.

---

14. Example Workflow for Pull Requests

Step 1: Fork and Clone the Repository

bash

Copy code

```
git clone https://github.com/original-owner/repository.git
cd repository
```

Step 2: Create a New Branch

bash
Copy code
git checkout -b fix-bug
Step 3: Make Changes and Commit
bash
Copy code
Edit files
git add .
git commit -m "Fix issue #123"
Step 4: Push the Changes
bash
Copy code
git push origin fix-bug
Step 5: Open a Pull Request

1. Go to your Git hosting platform.
2. Select your forked repository and click **Pull Requests**.
3. Click **New Pull Request**.
4. Choose the original repository's main branch as the base and your branch as the compare branch.
5. Add a description and submit the pull request.

10. To prevent pushing files to master branch

To prevent pushing files directly to the main branch in Git, you can implement a combination of **Git hosting platform settings** and **Git client-side configurations**. Here's how:

---

11. 1. Enforce Branch Protection Rules (On Git Hosting Platform)

Most Git hosting platforms like GitHub, GitLab, and Bitbucket offer branch protection rules to prevent direct pushes to the main branch.

On GitHub

1. Navigate to your repository on GitHub.
2. Go to **Settings** > **Branches**.
3. Under **Branch protection rules**, click **Add branch protection rule**.

4. Configure the rules:
    - **Branch name pattern**: Set to main.
    - Enable **Require pull request reviews before merging**.
    - Enable **Require status checks to pass before merging**.
    - Enable **Include administrators** (if applicable).
5. Save the branch protection rule.

On GitLab

1. Navigate to your repository on GitLab.
2. Go to **Settings** > **Repository** > **Protected Branches**.
3. Add the main branch to the **Protected Branches** list.
4. Set permissions:
    - **Allowed to merge**: Maintainers only.
    - **Allowed to push**: No one (or select specific roles/users).

On Bitbucket

1. Navigate to your repository on Bitbucket.
2. Go to **Repository settings** > **Branch permissions**.
3. Add a rule for the main branch:
    - Disable **Allow anyone to push to this branch**.
    - Enable **Require pull request approvals**.

---

12. 2. Use Pre-Push Hooks (Local Git Configuration)

You can use Git hooks to prevent accidental pushes to the main branch from your local machine.

Steps to Set Up a Pre-Push Hook

1. Navigate to the .git/hooks directory in your repository.
2. Create or edit the pre-push hook:

bash

Copy code

nano .git/hooks/pre-push

3. Add the following script to block pushes to main:

bash

Copy code

#!/bin/bash

```
branch="$(git rev-parse --abbrev-ref HEAD)"
if [ "$branch" = "main" ]; then
    echo "Pushing directly to the 'main' branch is not allowed."
    exit 1
fi
```

4. Save and make the hook executable:

```
bash
Copy code
chmod +x .git/hooks/pre-push
```

---

13. 3. Set Default Branch for Development

Encourage developers to work on a separate branch, like dev, by setting it as the default branch.

On GitHub:

1. Go to **Settings** > **Branches**.
2. Under **Default branch**, click **Change default branch**.
3. Select a branch like dev or develop.

---

14. 4. Educate Team Members

Ensure your team follows these practices:

- Always create a new branch for features or fixes.
- Use pull requests for merging changes into the main branch.
- Never force-push to main.

---

15. Example Workflow After Setup

1. **Developer Workflow**:
    - Create a branch:

```
bash
Copy code
git checkout -b feature-branch
```

    - Make changes, commit, and push:

```
bash
Copy code
git add .
```

git commit -m "Add feature"
git push origin feature-branch

- Open a pull request to merge changes into main.

2. **Repository Maintainer**:
   - Review and approve the pull request.
   - Merge the changes into the main branch.

11. Feature Branch and Hotfix Branch

In Git-based workflows, **feature branches** and **hotfix branches** are specialized types of branches used for different purposes. Here's a breakdown:

---

12. 1. Feature Branch

Purpose:

Feature branches are used to develop new features or enhancements for the project. They are typically created off the main development branch (e.g., main, develop) and merged back after the feature is completed and reviewed.

Key Characteristics:

- **Branch source**: Usually created from the develop branch in workflows like Git Flow.
- **Naming convention**: Use descriptive names like feature/feature-name or feat/login-page.
- **Lifecycle**: Short-lived; exists only until the feature is complete and merged.

Workflow:

1. **Create a Feature Branch**:

bash
Copy code
git checkout -b feature/feature-name develop

2. **Work on the Feature**:
   - Make changes, commit, and push:

bash
Copy code
git add .
git commit -m "Implement feature-name"

git push origin feature/feature-name

3. **Open a Pull Request**: Merge the feature branch into the develop branch after review and testing.
4. **Merge and Delete**:
    - Once approved, merge the pull request.
    - Optionally delete the feature branch:

bash
Copy code
git branch -d feature/feature-name

---

13. 2. Hotfix Branch

Purpose:

Hotfix branches are used to quickly address critical issues, such as bugs or vulnerabilities, that need to be resolved immediately. These changes are typically merged into both the main and develop branches to keep both production and development environments up-to-date.

Key Characteristics:

- **Branch source**: Created from the main branch to fix an issue affecting production.
- **Naming convention**: Use descriptive names like hotfix/hotfix-name or fix/critical-bug.
- **Lifecycle**: Short-lived; exists only until the fix is complete and merged.

Workflow:

1. **Create a Hotfix Branch**:

bash
Copy code
git checkout -b hotfix/hotfix-name main

2. **Apply the Fix**:
    - Make changes, commit, and push:

bash
Copy code
git add .
git commit -m "Fix critical issue hotfix-name"

git push origin hotfix/hotfix-name

3. **Merge the Hotfix**:
    - Merge the hotfix into the main branch for deployment:

bash
Copy code
git checkout main
git merge hotfix/hotfix-name
git push origin main

    - Merge the hotfix into the develop branch to keep development updated:

bash
Copy code
git checkout develop
git merge hotfix/hotfix-name
git push origin develop

4. **Delete the Hotfix Branch**:

bash
Copy code
git branch -d hotfix/hotfix-name

---

14. Comparison: Feature Branch vs. Hotfix Branch

Aspect	Feature Branch	Hotfix Branch
Purpose	Develop new features or enhancements	Fix critical issues in production
Source Branch	Usually develop	main
Merge Target	develop (or equivalent)	main and develop
Priority	Lower priority, depends on roadmap	High priority, immediate action
Naming Convention	feature/feature-name	hotfix/hotfix-name
Lifecycle	Exists until feature is complete	Short-lived, exists until fix is deployed

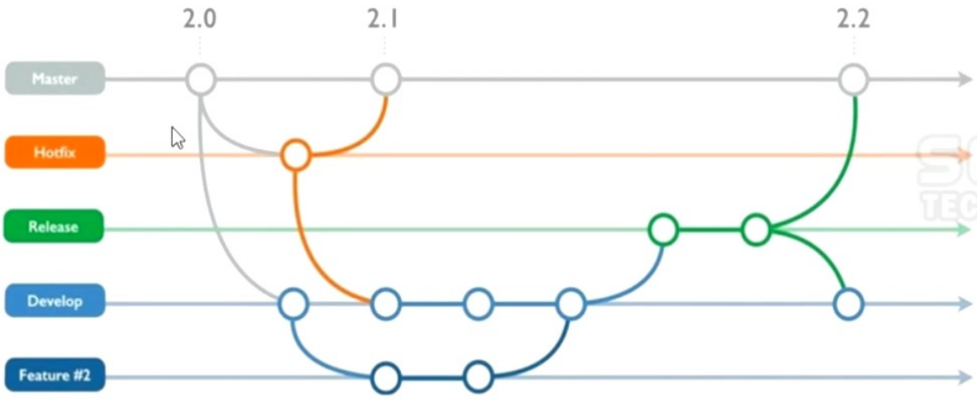
15. When to Use Each

- **Feature Branch**:
  - When working on new functionality, improvements, or experiments.
  - Example: Developing a new login feature or adding a report generation tool.
- **Hotfix Branch**:
  - When resolving urgent issues affecting production, such as security vulnerabilities, crashes, or data corruption.
  - Example: Fixing a typo causing a 500 error or updating an expired API key.

- To delete any in-between commits - git revert <revision id>
- To create branch – git branch <branch name>
- To switch branch – git checkout <target branch name>
- To delete branch – git branch -d <branch name>
- To rebase branch – git rebase <from branch> <target branch>
- To bring only specific commit – git cherry-pick <Revision id>
- To stash few changes – git stash
- To see all stash list – git stash list
- To bring stashed changes back – git stash pop <pop id (optional)>
- To squash many commits into single commit – squash is used

4. Create branch, then files and merge with GitHub Repository for Individual

Git Branching Strategy and Workflow Examples



Please connect with me via below social media

Instagram: https://www.instagram.com/rba_infotech/
LinkedIn:
https://www.linkedin.com/company/70977316/admin/dashboard/
Facebook:https://www.facebook.com/people/RBA-Infotech/100043552406579/

```
# Step 1: Create a new branch
git checkout -b <branch-name>

# Step 2: Create and edit files
touch newfile.txt
# (Edit the file in your text editor)

# Step 3: Stage and commit changes
git add newfile.txt
git commit -m "Add newfile.txt with initial content"

# Step 4: Push the branch to remote
git push origin <branch-name>

# Step 5: Merge the branch into the main branch
git checkout main
git pull origin main
git merge <branch-name>
git push origin main
```

git config --list

5. Branching

Branches in GitHub serve as a powerful tool for managing different versions of a project, enabling teams and individual developers to work on new features, bug fixes, or experiments in isolation from the main codebase. Here's a breakdown of the purpose and benefits of using branches in GitHub:

Purpose of Branches in GitHub

1. **Isolated Development**:
   - **Purpose**: Branches allow developers to work on new features, fixes, or other changes without affecting the main codebase

(often referred to as the main or master branch).
    - **Example**: A developer can create a branch called feature/login to implement a new login feature. This keeps the main branch stable and clean while the new feature is being developed.
2. **Parallel Development**:
    - **Purpose**: Multiple developers or teams can work on different features or fixes simultaneously, each in their own branch. This enables parallel development and faster progress.
    - **Example**: While one developer is working on feature/login, another might be working on feature/signup, both in separate branches.
3. **Version Control**:
    - **Purpose**: Branches help manage different versions of a project. For example, a release branch might be used for preparing a new version of the software, while the main branch continues to evolve with new features.
    - **Example**: A v1.0 branch could be used to maintain the first stable release of a project, while ongoing development happens in main.
4. **Safe Experimentation**:
    - **Purpose**: Developers can create branches to experiment with new ideas, test new technologies, or try out changes without any risk to the main codebase.
    - **Example**: A developer might create an experiment/try-new-api branch to test integrating a new API. If it doesn't work out, the branch can be deleted without any impact on the rest of the project.
5. **Code Review and Collaboration**:
    - **Purpose**: Branches are integral to the code review process. Developers can submit a pull request from their branch, allowing others to review the changes before they are merged into the main branch.
    - **Example**: After completing work on feature/login, a developer can open a pull request for that branch. Team members can review the changes, suggest improvements, and finally merge it into main after approval.
6. **Continuous Integration and Deployment (CI/CD)**:
    - **Purpose**: Branches play a key role in CI/CD pipelines, where automated tests and deployments are triggered on specific branches (e.g., main, develop).
    - **Example**: Code pushed to the development branch might trigger a build process and run tests, ensuring that everything

is stable before merging into the main branch for deployment.

7. **Hotfixes**:
    - **Purpose**: When an urgent issue arises in production, a hotfix branch can be created from the main branch to quickly address the problem. Once the fix is made, the branch can be merged back into the main branch and deleted.
    - **Example**: If a critical bug is found in production, a hotfix/security-patch branch can be created to fix it. After testing, the fix can be merged back into main and deployed.

Summary

Branches in GitHub are essential for managing changes in a project. They allow for isolated and parallel development, facilitate collaboration through pull requests, support CI/CD processes, and provide a safe environment for experimentation. By using branches effectively, teams can maintain a clean and stable main codebase while continuously developing and improving their projects

6. Git Command - Push

The `git push` command is used to upload your local repository content to a remote repository. When you push, you are sending your committed changes to the remote repository, typically to update a branch on the remote.

Basic Syntax:
bash
git push <remote_name> <branch_name>

7. Key Components:

<remote_name>: The name of the remote repository you are pushing to (e.g., `origin` is the default name for the remote repository when you clone).

<branch_name>: The name of the branch you want to push. Typically, this could be `main`, `master`, or any other branch you're working on.

8. Example Command:

bash
git push origin main

This command pushes the commits from your local `main` branch to the `main` branch on the remote repository named `origin`.

9. Common Options:

git push -u <remote_name> <branch_name>: Sets the upstream branch for the local branch, so you can use `git push` without specifying the remote and branch names in the future.

bash
git push -u origin main

git push --force` or `git push -f: Forces the push even if it will overwrite changes in the remote branch. This is useful if you need to rewrite history, but should be used with caution as it can cause issues for collaborators.

bash
git push -f origin main

git push --all: Pushes all branches to the remote repository.

bash
git push --all origin

git push --tags: Pushes all tags to the remote repository. Tags are used to mark specific points in history, such as releases.

bash
git push origin --tags

10. Common errors with the `git push` command:

1. ! [rejected]: Local branch is out of sync with remote.
Solution: Use `git pull --rebase` to sync.

2. error: failed to push some refs: Local and remote branches have diverged.
Solution: Use `git pull --rebase`.

3. Permission denied (publickey): SSH key not found or not associated with your Git account.
Solution: Add the correct SSH key.

4. remote: Repository not found. : The repo doesn't exist or you lack permissions.
Solution: Check the remote URL and your access rights.

5. fatal: remote origin already exists.: Attempting to add a remote that already exists.
Solution: Use `git remote set-url` to update the remote.

6. fatal: Not a git repository: Running Git commands outside a Git repo.
Solution: Navigate to the correct directory.

7. Updates were rejected: Local branch is behind the remote.
Solution: Use `git pull --rebase`.

8. fatal: The current branch has no upstream branch. : No upstream branch is set for the current branch.
Solution: Use `git push --set-upstream origin <branch>`.

9. SSL certificate problem: SSL certificate validation issue.
Solution: Update CA certificates or disable SSL verification.

10. Could not resolve host: DNS issues.
Solution: Check your internet connection and DNS settings.

These are the most common errors you might encounter when pushing changes with Git.

11. Workflow Context:

Typically, the `git push` command is used after you've made changes, committed those changes locally with `git commit`, and are ready to share those changes with others by pushing them to the remote repository.

12. Example Workflow:

1. Make changes to your files.
2. Stage changes: `git add .`
3. Commit changes: `git commit -m "Description of changes"`
4. Push changes: `git push origin main`

This workflow ensures that your local changes are reflected in the remote repository, making them accessible to others.

13. Git Command - Pull

The `git pull` command is used to fetch and integrate changes from a remote repository into your local repository. It combines two operations: `git fetch` (which downloads changes from the remote repository) and `git merge` (which integrates those changes into your current branch).

Basic Syntax:
bash
git pull <remote_name> <branch_name>

14. Key Components:

<remote_name>: The name of the remote repository (e.g., `origin` is the default name for the remote repository when you clone).

<branch_name>: The name of the branch you want to pull the changes from. Typically, this could be `main`, `master`, or any other branch you're working with.

15. Example Command:

bash
git pull origin main

This command fetches the latest changes from the `main` branch of the remote repository named `origin` and merges them into your current local branch.

16. Common Scenarios:

Default Pull: If your local branch is already tracking a remote branch (set by commands like `git push -u`), you can simply use:

bash
git pull

Fast-forward Merge: If your local branch is behind the remote branch but has no divergent commits, `git pull` will perform a fast-forward merge, effectively updating your local branch without creating a merge commit.

Merge Conflicts: If there are conflicting changes between your local commits and the remote changes, `git pull` will trigger a merge conflict. You'll need to manually resolve these conflicts, stage the resolved files, and commit the merge.

17. common errors with the `git pull` command:

1. CONFLICT (content): Merge conflict : Local changes conflict with remote changes.
Solution: Resolve the conflicts manually, then commit.

2. fatal: refusing to merge unrelated histories: Repositories have no common base commit.
Solution: Use `git pull --allow-unrelated-histories`.

3. error: Your local changes to the following files would be overwritten by merge: Uncommitted local changes conflict with remote updates.
Solution: Commit or stash local changes before pulling.

4. fatal: Not a git repository : Running `git pull` outside a Git repository.
Solution: Ensure you're in a valid Git directory.

5. fatal: Could not read from remote repository: Issues with remote access, often due to SSH or authentication problems.
Solution: Check SSH key and remote repository URL.

6. error: Pulling is not possible because you have unmerged files: Unresolved merge conflicts from a previous merge.
Solution: Resolve the conflicts, then commit and pull again.

7. fatal: The current branch <branch-name> has no upstream branch.: No upstream branch is set for the current branch.
Solution: Use `git branch --set-upstream-to origin/<branch>` to set it.

8. error: cannot lock ref <ref>: Another process is holding a lock on the repository.
Solution: Wait for the process to finish or delete the lock file if it's safe.

9. fatal: unable to access 'https://<repo-url>/': Could not resolve host: DNS or network issues prevent accessing the remote.
Solution: Check your network connection and DNS settings.

10. error: Server does not allow request for unadvertised object: The remote server does not advertise all objects, often due to shallow clones.
Solution: Fetch all branches and tags with `git fetch --all`.

These are some of the most common issues you might face when using `git pull`.

18. Example Workflow:

1. Ensure you're on the correct branch: `git checkout main`
2. Pull the latest changes: `git pull origin main`
3. Resolve any merge conflicts (if any).
4. Test your code to ensure everything works as expected after the merge.

git pull --rebase: Instead of merging the fetched changes, this option rebases your local changes on top of the fetched changes. This results in a linear history without merge commits.

bash
```
git pull --rebase origin main
```

19. Workflow Context:

Use `git pull` regularly to keep your local repository in sync with the remote repository, especially when collaborating with others. This ensures

you have the latest code before making additional changes.

20. Git Clone

The `git clone` command is used to create a copy of an existing remote Git repository on your local machine. When you clone a repository, you download all the files, history, branches, and configuration from the remote repository.

Basic Syntax:
bash
git clone <repository_url>

21. Key Components:

<repository_url>: The URL of the remote repository you want to clone. This can be an HTTPS URL, SSH URL, or a local path.

22. Example Command:

bash
git clone https://github.com/user/repository.git

This command clones the repository from the specified URL into a new directory on your local machine named `repository`.

23. Common Options:

git clone <repository_url> <directory_name>: Clones the repository into a specific directory. If you don't specify a directory name, Git will create one based on the repository name.

bash
git clone https://github.com/user/repository.git my-directory

git clone --branch <branch_name> <repository_url>: Clones a specific branch from the remote repository, rather than the default branch (usually `main` or `master`).

bash
git clone --branch develop https://github.com/user/repository.git

git clone --depth <depth> <repository_url>: Creates a shallow clone with a limited history, which is useful if you don't need the entire history and want to save space.

bash
git clone --depth 1 https://github.com/user/repository.git

Common errors with the `git clone` command:

1. fatal: repository not found: The repository URL is incorrect or the repo doesn’t exist.
Solution: Verify the URL and your access permissions.

2. Permission denied (publickey): SSH key not found or not authorized.
Solution: Add the correct SSH key to your SSH agent and Git account.

3. fatal: Could not read from remote repository: Authentication issues or SSH problems.
Solution: Check your SSH configuration and remote URL.

4. SSL certificate problem: unable to get local issuer certificate : SSL certificate validation issue.
Solution: Update your CA certificates or disable SSL verification.

5. fatal: destination path '<directory>' already exists and is not an empty directory: The target directory already exists and isn’t empty.
Solution: Choose a different directory or remove the existing one.

6. error: RPC failed; curl 18 transfer closed with outstanding read data remaining: Network issues during clone.
Solution: Try cloning again, or use a different network.

7. warning: You appear to have cloned an empty repository. : The repository is empty.
Solution: Ensure that this is expected. No action needed if it's correct.

8. fatal: unable to access 'https://<repo-url>/': Could not resolve host : DNS or network issues prevent accessing the remote.
Solution: Check your network connection and DNS settings.

9. fatal: remote end hung up unexpectedly : Network disruption during clone.
Solution: Try cloning again, or use a more stable network.

10. error: Server does not allow request for unadvertised object : Attempting to clone a specific branch or commit that isn't advertised by the server.
Solution: Clone the entire repository or ensure the branch/commit is available.

These are the common errors you might encounter when using `git clone`.

24. Workflow Context:

Initial Setup: You typically use `git clone` when you first start working on an existing project. It sets up a local repository that is linked to the remote repository, enabling you to work on the project locally.

Collaboration: Cloning is also the first step for contributors who want to work on an open-source project or a shared repository.

25. Example Workflow:

1. Clone the Repository:
bash
git clone https://github.com/user/repository.git

2. Navigate into the Cloned Directory:
bash
cd repository

3. Start Working: You can now create branches, make changes, commit, push, and pull from this local copy of the repository.

The `git clone` command is essential for initializing your work on an existing project by creating a complete copy on your local machine.

26. Git Command - Branching

Branching in Git allows you to create separate lines of development within

a repository. This is useful for working on features, fixing bugs, or experimenting without affecting the main codebase.

27. Key Branching Commands:

1. Creating a New Branch:
Command: `git branch <branch_name>`
Description: Creates a new branch with the specified name. This branch will be based on the current branch's latest commit.

bash
git branch feature-branch

2. Switching to a Branch:
Command : `git checkout <branch_name>`
Description: Switches to the specified branch, making it the current branch. Your working directory will be updated to reflect the files and state of that branch.

bash
git checkout feature-branch

Shortcut for Creating and Switching:
Command: `git checkout -b <branch_name>`
Description: Creates a new branch and immediately switches to it.
bash
git checkout -b new-feature

3. Listing All Branches:
Command: `git branch`
Description: Lists all branches in the repository. The current branch is highlighted with an asterisk (`*`).

bash
git branch

4. Deleting a Branch:
Command: `git branch -d <branch_name>`
Description: Deletes the specified branch. This is a safe delete and will only delete the branch if it has been fully merged.

bash
git branch -d feature-branch

Force Delete:
Command: `git branch -D <branch_name>`
Description: Forcefully deletes the branch, even if it has not been merged.

bash
git branch -D feature-branch

28. Common errors with Git branching commands:

1. fatal: A branch named '<branch-name>' already exists. : Trying to create a branch that already exists.
Solution: Use a different branch name or switch to the existing branch with `git checkout <branch-name>`.

2. error: The branch '<branch-name>' is not fully merged.: Trying to delete a branch that hasn't been merged.
Solution: Merge the branch first, or force delete with `git branch -D <branch-name>`.

3. fatal: Not a valid object name: '<branch-name>': The branch name doesn't exist.
Solution: Check for typos in the branch name or ensure the branch was created.

4. fatal: Cannot delete branch '<branch-name>' checked out at '<directory>': Attempting to delete the branch you're currently on.
Solution: Switch to a different branch with `git checkout <another-branch>` and then delete.

5. fatal: branch '<branch-name>' has multiple upstream branches, refusing to guess: The branch is tracking multiple upstream branches.
Solution: Specify the correct upstream branch or fix the tracking with `git branch --set-upstream-to`.

6. error: refname refs/heads/<branch-name> not found: Trying to check out a non-existent branch.

Solution: Ensure the branch name is correct or create it if needed.

7. error: failed to push some refs: Issues pushing a branch to a remote, often due to lack of tracking or conflicts.
Solution: Set the upstream branch with `git push --set-upstream origin <branch-name>` or resolve conflicts.

8. fatal: A branch named 'HEAD' cannot be created: Attempting to name a branch "HEAD", which is reserved.
Solution: Choose a different branch name.

9. error: Cannot rename branch '<branch-name>' in a detached HEAD state: Trying to rename a branch while in a detached HEAD state.
Solution: Check out a branch first with `git checkout <branch-name>`.

10.error: failed to create symbolic link: Issues with symbolic links when creating or checking out a branch.
Solution: Check for permissions or file system compatibility with symbolic links.

These are the common issues you might encounter when working with Git branches.

29. Creating branch and pushing it to remote

1. Create a New Branch:
bash
git branch feature-login

2. Switch to the New Branch:
bash
git checkout feature-login

3. Make Changes and Commit:
Make your changes to the code.
Stage and commit the changes.
bash
git add .
git commit -m "Added login feature"

4. Push the Branch to a Remote (optional):

bash
git push origin feature-login

5. Merge Back to Main Branch:
After completing work, switch back to the `main` branch and merge your changes.
bash
git checkout main
git merge feature-login

6. Delete the Branch (if no longer needed):
bash
git branch -d feature-login

30. Workflow Context:

Feature Development: Use branches to develop new features without affecting the main codebase.
Bug Fixes: Create a branch for fixing bugs to keep the main branch stable.
Experimentation: Try out new ideas in a branch without risking the stable code.

Branching allows for organized, parallel development, making it easier to manage large projects and collaborate with others.

31. Git Command - Merging

Merging in Git is the process of integrating changes from one branch into another. This is commonly done to bring the changes from a feature branch into the main branch after development is complete.

32. Key Merging Commands:

1. Merging a Branch:
Command: `git merge <branch_name>`
Description: Merges the specified branch into the current branch. Git tries to automatically combine the changes from both branches.

Bash
git checkout main
git merge feature-branch

In this example, the changes from `feature-branch` are merged into the `main` branch.

2. Fast-Forward Merge:
Description: If the current branch has not diverged from the branch being merged, Git performs a fast-forward merge, simply moving the branch pointer forward.

Example (automatically done by `git merge` if conditions allow):
bash
git merge feature-branch

3. Three-Way Merge:
Description: If the branches have diverged, Git performs a three-way merge, combining the changes and creating a new merge commit to record the merge.

Example:
bash
git merge feature-branch

4. Handling Merge Conflicts:
Description: If there are conflicting changes in the same parts of the files, Git will pause the merge and mark the conflicting files.
Steps to Resolve Conflicts:

1. Open the conflicting files in a text editor.
2. Look for conflict markers (`<<<<<<<`, `=======`, `>>>>>>>`) and resolve the conflicts by choosing or combining the changes.
3. Stage the resolved files with `git add`.
4. Complete the merge by committing the changes.

bash
```
git add resolved-file.txt
git commit -m "Resolved merge conflict
```

5. Aborting a Merge:
Command: `git merge --abort`
Description: Aborts the merge process and returns the branch to its state before the merge began. Useful if you want to cancel the merge due to conflicts or other issues.

bash
```
git merge –abort
```

33. Common errors with Git merging commands:

1. CONFLICT (content) : Merge conflicts in files.
Solution: Manually resolve conflicts in the affected files, then stage and commit the changes.

2. error: Merging is not possible because you have unmerged files: Conflicts from a previous merge are unresolved.
Solution: Resolve the conflicts and complete the merge.

3. fatal: refusing to merge unrelated histories: Repositories have no common base commit.
Solution: Use `git merge --allow-unrelated-histories`.

4. fatal: Not possible to fast-forward, aborting: Merge requires a non-fast-forward update but is set to fast-forward only.
Solution: Use `git merge --no-ff` to force a merge commit.

5. fatal: You have uncommitted changes: Local changes prevent merging.
Solution: Commit or stash your changes before merging.

6. error: Your local changes to the following files would be overwritten by merge: Local changes would be lost by the merge.
Solution: Commit or stash local changes before merging.

7. fatal: cannot merge branch '<branch-name>' into '<branch-name>':

Issues merging specific branches, possibly due to conflicts or errors.
Solution: Investigate conflicts or errors and resolve them.

8. error: The following untracked working tree files would be overwritten by merge:: Untracked files would be overwritten.
Solution: Add or remove untracked files as needed, or stash them.

9. fatal: reference is not a tree: <commit-hash>: Trying to merge a non-existent commit.
Solution: Verify the commit hash or branch name.

10. fatal: Cannot merge; untracked working tree files present: Untracked files prevent merging.
Solution: Add or remove untracked files, or stash them.

These are common issues you might face when merging in Git.

34. Example Workflow:

1. Switch to the Target Branch (e.g., `main`):
bash
git checkout main

2. Merge the Feature Branch :
bash
git merge feature-branch

3. Resolve Conflicts (if any):
Edit the files to resolve conflicts.
Stage the resolved files.
Complete the merge.

4. Finish the Merge:
bash
git commit -m "Merged feature-branch into main"

35. Workflow Context:

1. Integrating Features: Merge feature branches into the main branch

after development and testing are complete.

2. Syncing Branches: Merge changes from the main branch into a feature branch to keep it up-to-date with the latest changes.
3. Collaboration: Merging allows different team members to combine their work, ensuring everyone's changes are integrated into the project.

Merging is a crucial part of Git workflows, allowing parallel development and easy integration of changes from multiple branches.

36. Git clone vs git pull

Git clone	Git pull
Used to create a local copy of a remote repository	Used to update your local repository with changes from the remote
Copies the entire repository history and all branches	Fetches the remote branches and merges the changes into your current local branch
Typically done only once at the beginning of a project	Should be done regularly to keep your local repository up-to-date
Example: git clone https://github.com/user/repo.git	Example: git pull origin main

37. Git fetch vs git pull

Feature	git fetch	git pull
Operation	Retrieves changes without merging	Retrieves and merges changes
Impact on Local Files	No changes to local files or branches	Updates local files and branches
Use Case	Review changes before merging	Quick synchronization with remote
Command Syntax	git fetch [remote] [branch]	git pull [remote] [branch]

Feature	git fetch	git pull
Risk of Conflicts	Lower risk, allows for manual merging	Higher risk, may lead to merge conflicts

38. Various error occurred in GitHub commands and solutions

1. not a git repository (or any of the parent directories): .git

Cause: Running a Git command in a non-Git directory.
Solution: Navigate to a Git repository or initialize one with `git init`.

2. remote origin already exists.

Cause: Attempting to add a remote named `origin` that already exists.
Solution: Use `git remote set-url origin <URL>` to update the remote URL or remove it with `git remote remove origin`.

1. failed to push some refs

   Cause: Local branch is behind the remote branch.
   Solution: Use `git pull --rebase origin <branch>` to update your local branch, then push again. Alternatively, force push with `git push -f` (use with caution).

2. refusing to merge unrelated histories

   Cause: Attempting to merge two repositories with no common history.
   Solution: Use `git merge <branch> --allow-unrelated-histories`.

3. Permission denied (publickey)

   Cause: SSH key authentication issue.
   Solution: Add your SSH key to the SSH agent with `ssh-add ~/.ssh/id_rsa`, and ensure it's added to your GitHub account.

4. Your local changes to the following files would be overwritten by merge
   Cause: Local changes conflict with incoming changes.
   Solution: Stash your changes with `git stash`, pull the latest changes, and apply the stash with `git stash pop`.

5. could not read Username for 'https://github.com': terminal prompts disabled
   Cause: Git is trying to prompt for a username in a non-interactive environment.
   Solution: Use a Git credential helper or switch to SSH by running `git remote set-url origin git@github.com:<username>/<repository>.git`.

6. failed to push some refs to '<repository>'
   Cause: Local repository has diverged from the remote.
   Solution: Run `git pull --rebase` and resolve any conflicts before pushing.

7. CONFLICT (content): Merge conflict in <filename>
   Cause: A merge conflict where the same part of a file has been modified in different ways.
   Solution: Manually resolve the conflicts, then use `git add <filename>` and `git commit`.

8. unable to auto-detect email address
   Cause: Git is unable to find an email address in your configuration.
   Solution: Set your email with `git config --global user.email "your_email@example.com"` and your name with `git config --global user.name "Your Name"`.

9. RPC failed; curl 56 GnuTLS recv error (-9)
   Cause: Often related to large files or slow network connections.
   Solution: Increase the buffer size with `git config --global http.postBuffer 524288000` and retry.

10. cannot lock ref 'refs/remotes/origin/branch': is at...
    Cause: Race condition during a push or pull operation.
    Solution: Clean up stale references with `git remote prune origin`.

Please connect with me via below social media

Instagram: https://www.instagram.com/rba_infotech/
LinkedIn:
https://www.linkedin.com/company/70977316/admin/dashboard/
Facebook:https://www.facebook.com/people/RBA-Infotech/100043552406579/

11. The following untracked working tree files would be overwritten by checkout:
    Cause: Untracked files will be overwritten by checkout.
    Solution: Remove untracked files with `git clean -f` or back them up manually.

12. repository '<URL>' not found
    Cause: The repository URL is incorrect or does not exist.
    Solution: Verify the repository URL or clone the correct one.

13. unable to connect to <hostname>
    Cause: Network connection issues or incorrect remote URL.
    Solution: Check your network connection and verify the remote URL.

14. detected dubious ownership in repository
    Cause: Ownership mismatch of files within the repository.
    Solution: Change the ownership or suppress the warning with `git config --global --add safe.directory /path/to/repository`.

15. unable to create file (Filename too long)
    Cause: File path exceeds the OS limit.
    Solution: Enable long paths in Git by running `git config --system core.longpaths true` on Windows.

16. The current branch <branch> has no upstream branch.
    Cause: Pushing a branch without an upstream branch set.
    Solution: Set the upstream branch with `git push --set-upstream origin <branch>`.

17. ambiguous argument 'HEAD': unknown revision or path not in the working tree.
    Cause: The HEAD reference is invalid or corrupted.
    Solution: Rebuild the repository or reset the branch with `git reset --hard HEAD`.

18. branch '<branch>' is not fully merged.
    Cause: Attempting to delete a branch that is not fully merged.
    Solution: Force delete the branch with `git branch -D <branch>`, or merge it before deleting.

19. unable to access '<repository URL>': SSL certificate problem

Cause: Issues with SSL certificate verification.
Solution: Disable SSL verification (not recommended) with `git config --global http.sslVerify false` or fix the certificate issue.

20. unable to update url base from redirection
Cause: Git is unable to follow the redirect.
Solution: Update the remote URL to the correct one with `git remote set-url origin <new_url>`.

21. cannot create directory at '...': Permission denied
Cause: Insufficient permissions to create a directory or file.
Solution: Change directory permissions with `chmod` or use `sudo` to run the command as a superuser.

22. The requested URL returned error: 403 Forbidden while accessing <repository>
Cause: Insufficient permissions to access the repository.
Solution: Ensure you have the correct access rights or ask the repository owner to grant access.

23. couldn't find remote ref <branch>
Cause: The specified branch does not exist in the remote repository.
Solution: Verify the branch name and ensure it exists on the remote.

24. bad index file sha1 signature
Cause: Corruption in the index file.
Solution: Rebuild the index file with `rm -f .git/index` and `git reset`.

25. reference is not a tree
Cause: Trying to check out a branch or commit that doesn't exist.
Solution: Verify the commit or branch exists and is spelled correctly.

26. 'origin' does not appear to be a git repository
Cause: The remote origin is incorrectly set or does not exist.
Solution: Set the correct remote origin with `git remote add origin <repository_url>`.

27. index file is too large

Cause: The index file has grown too large.
Solution: Use Git LFS (Large File Storage) for handling large files, or split the repository.

28. couldn't read <filename>
Cause: File is missing or has been deleted.
Solution: Restore the file from a backup or remove it from the index with `git rm --cached <filename>`.

29. Git Remote Origin Not Found
Cause: You haven't set the remote origin. git remote -v or git fetch origin returns an error.
Solution: Run git remote add origin <repository - url> to set the remote origin.

30. Git Commit Failed
Cause: Modification aren't committed before pushing, there are Git pre-push hook difficulties,or the branch name is incorrect.
Solution: Check if you have write permissions, and ensure the file is not already committed.

31. Git Add All Files Not Working
Cause: This is because the node_modules folder is not ignored and hence when we run git add . command git visit all your directory/files and sub directory/files which is not ignored in .gitignore hence it is taking so long time to process it as node_modules contains a large amount of files and folder.
Solution: Run `git add -A` to stage all changes.

32. Git Reset Failed
Cause: If there could be conflicts between the changes in the commit we want to remove and the changes in the working tree we want to keep, the reset is disallowed
Solution: Run `git reset --hard` to reset the repository to the last commit.

33. Git Checkout Failed
Cause: If local modifications in a submodule would be overwritten the checkout will fail unless -f is used. If nothing (or --no-recurse-submodules) is used, submodules working trees will not be updated. Just like git-submodule[1], this will detach HEAD of the submodule.

Solution: Run `git checkout -b branch-name` to create a new branch.

34. Git Submodule Not Found

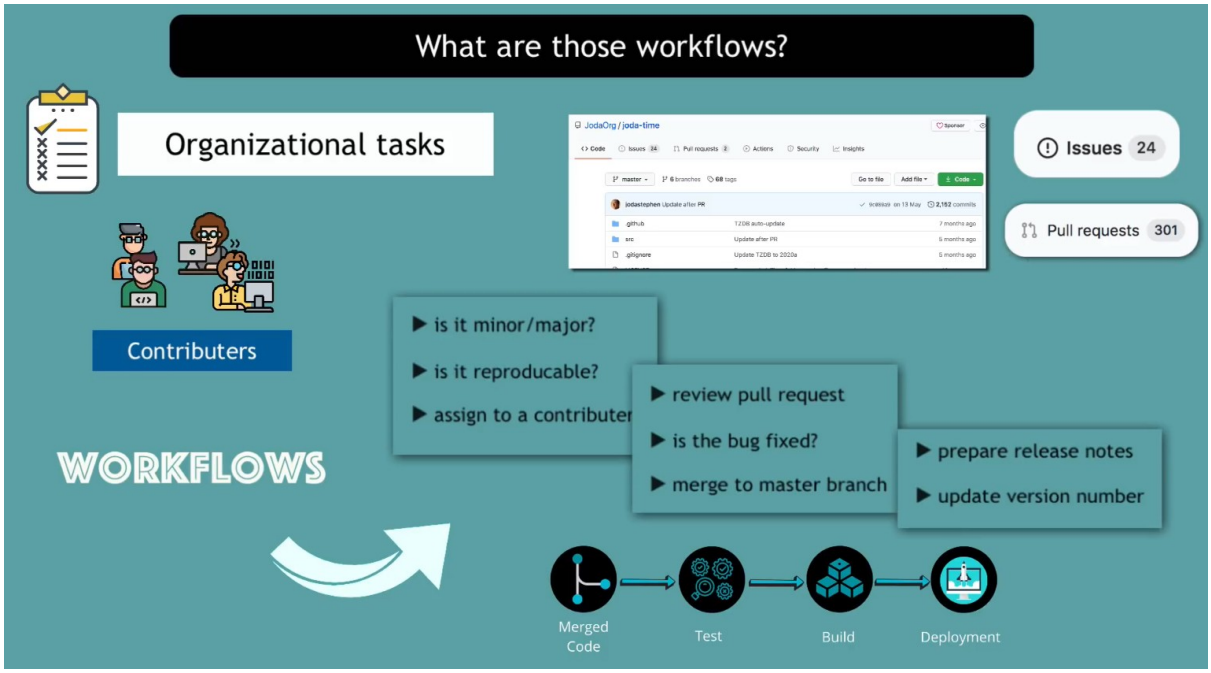
Cause: The root cause of the issue is that either the old submodule path remains cached (and must be be removed) or a path does not exist (and must be created) before the submodules may be updated.

Solution: Run `git submodule init` and `git submodule update` to initialize and update submodules.

3. GitHub Actions

- It is a platform to automate developer workflow
- CI/CD is one of the workflows

What is workflow?

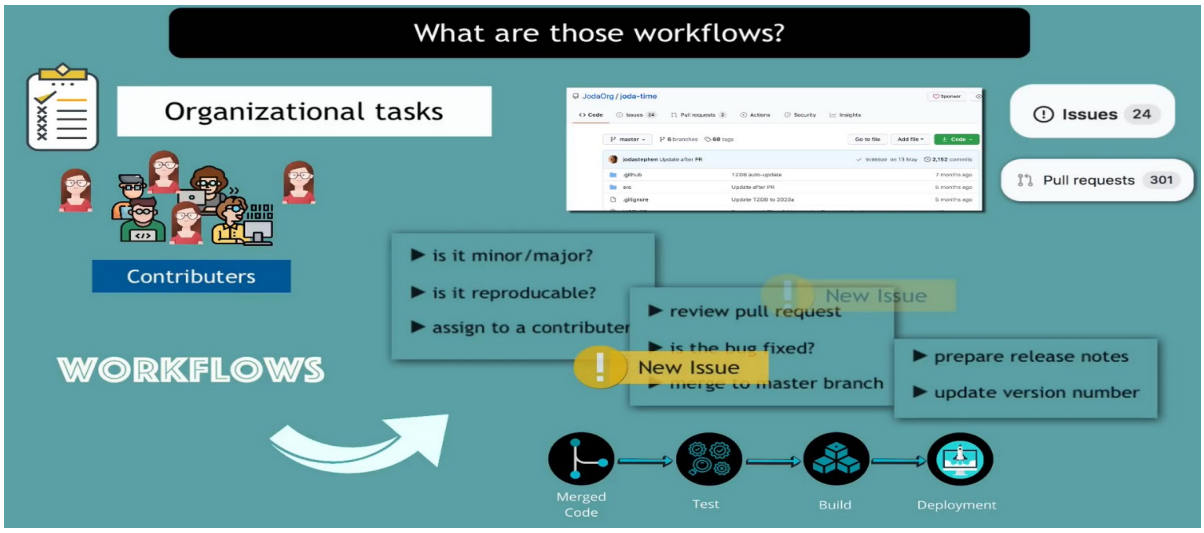


Please connect with me via below social media

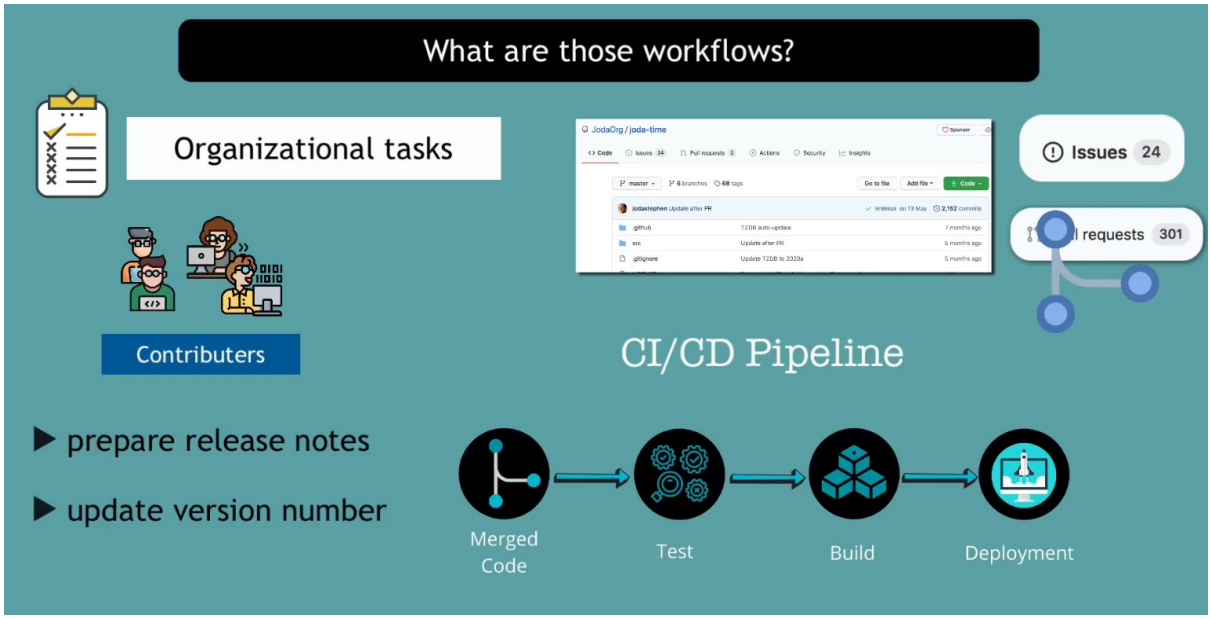
Instagram: https://www.instagram.com/rba_infotech/

LinkedIn: https://www.linkedin.com/company/70977316/admin/dashboard/

Facebook:https://www.facebook.com/people/RBA-Infotech/100043552406579/



How GitHub Actions automate the workflow

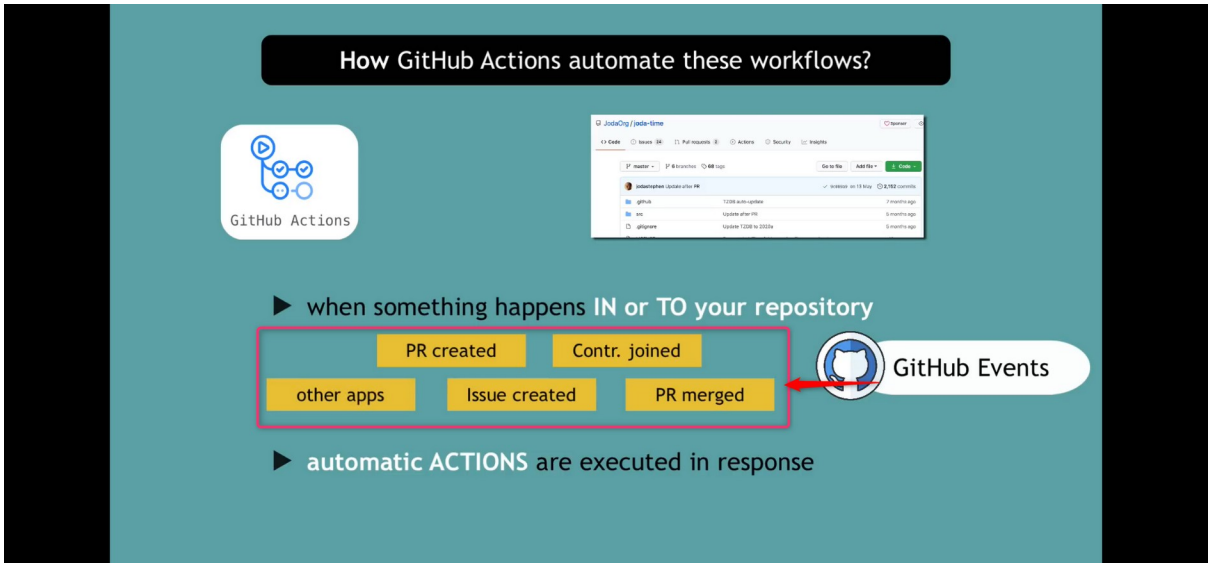


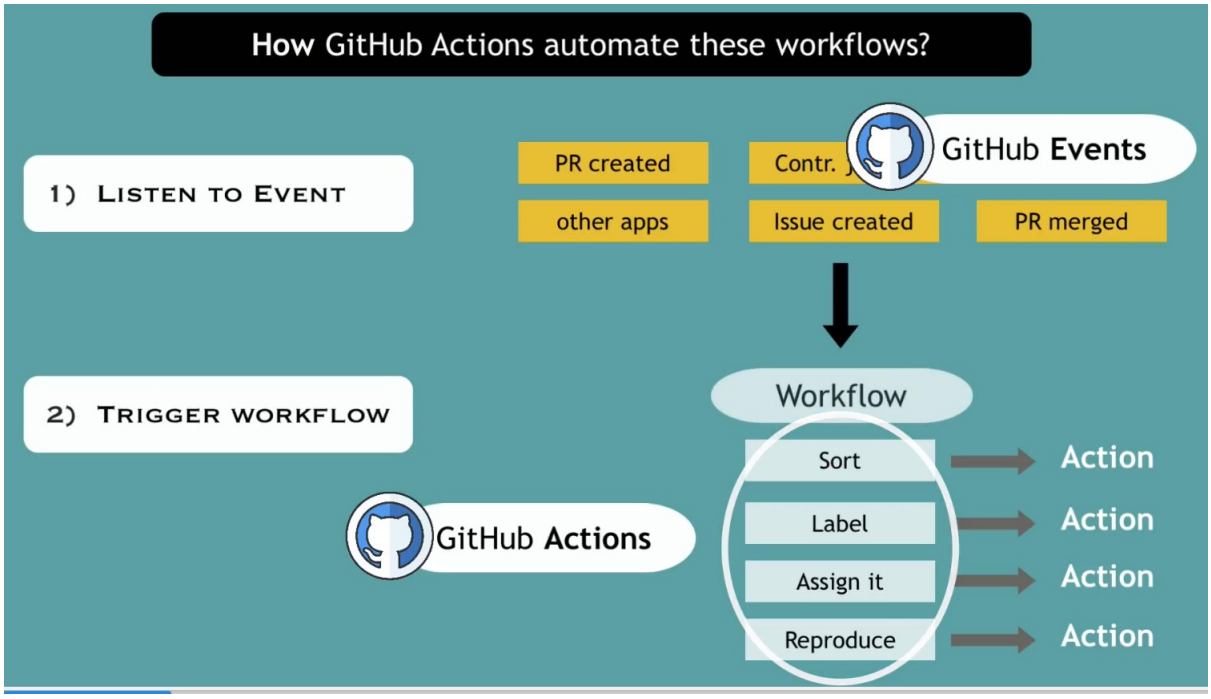
Please connect with me via below social media

Instagram: https://www.instagram.com/rba_infotech/

LinkedIn: https://www.linkedin.com/company/70977316/admin/dashboard/

Facebook:https://www.facebook.com/people/RBA-Infotech/100043552406579/





Please connect with me via below social media

Instagram: https://www.instagram.com/rba_infotech/

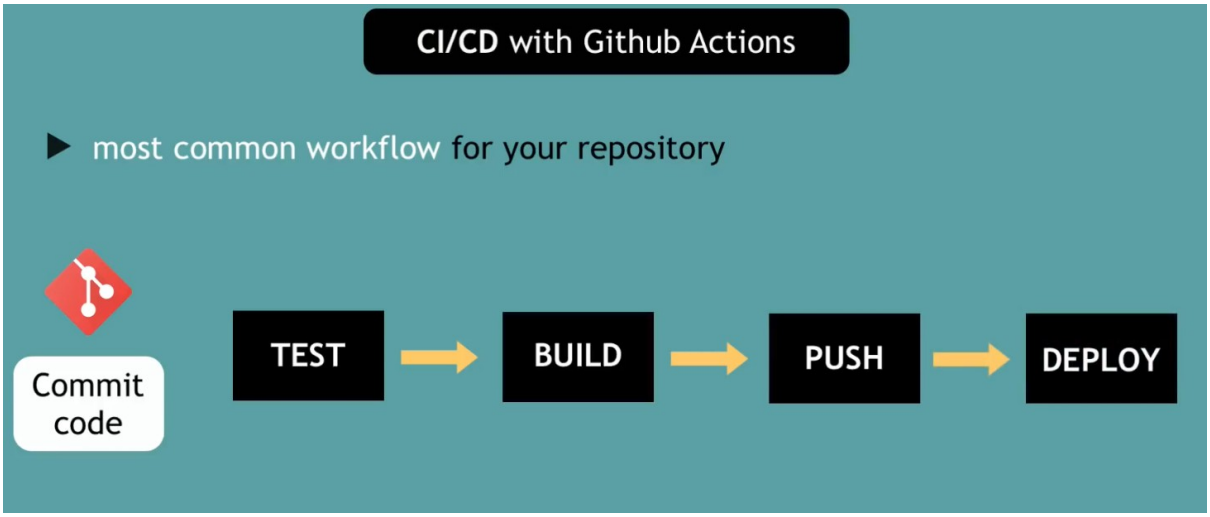
LinkedIn:
https://www.linkedin.com/company/70977316/admin/dashboard/

Facebook:https://www.facebook.com/people/RBA-Infotech/100043552406579/

Components of GitHub Actions

Below are the components of GitHub Actions

- Workflows
- Events
- Jobs
- Actions
- Runners

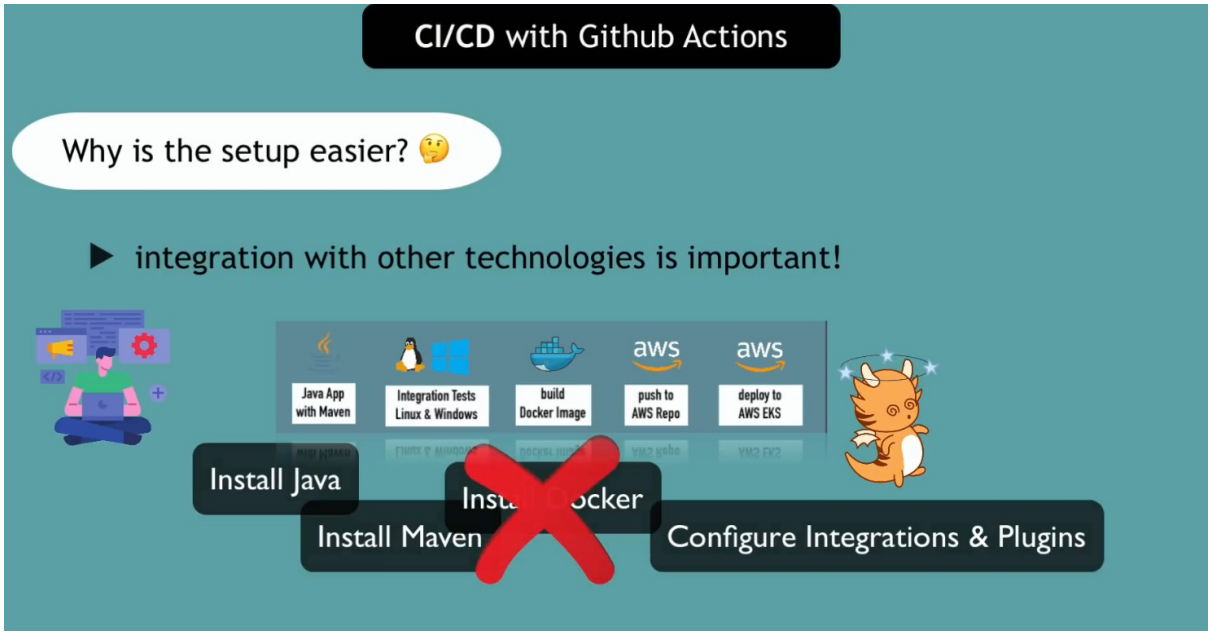


Why GitHub tool for CI/CD workflow?

- Use same tool instead of third party tool integration
- Setup the pipeline is easy
- Developer itself can setup this pipeline, no need of DevOps person to do this

Please connect with me via below social media

Instagram: https://www.instagram.com/rba_infotech/
LinkedIn:
https://www.linkedin.com/company/70977316/admin/dashboard/
Facebook:https://www.facebook.com/people/RBA-Infotech/100043552406579/

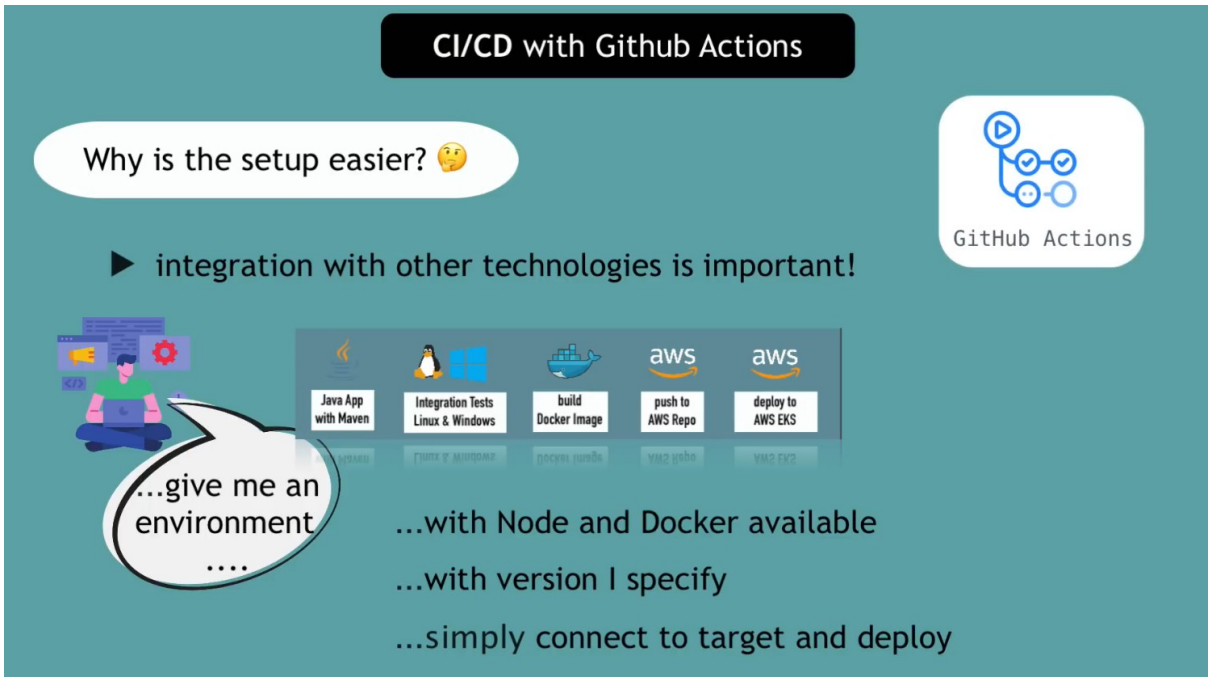


Please connect with me via below social media

Instagram: https://www.instagram.com/rba_infotech/

LinkedIn:
https://www.linkedin.com/company/70977316/admin/dashboard/

Facebook:https://www.facebook.com/people/RBA-Infotech/100043552406579/



Please connect with me via below social media

Instagram: https://www.instagram.com/rba_infotech/
LinkedIn:
https://www.linkedin.com/company/70977316/admin/dashboard/
Facebook:https://www.facebook.com/people/RBA-Infotech/100043552406579/